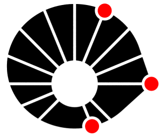

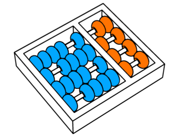

Um estudo comparativo entre sistemas auto-distributivos e serverless computing

A. L. C. Silva *G. P. Cardenete* *L. F. Bittencourt*
R. R. Filho

Technical Report - IC-PFG-23-30 - Relatório Técnico
December - 2023 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

The contents of this report are the sole responsibility of the authors.
O conteúdo deste relatório é de única responsabilidade dos autores.

Um estudo comparativo entre sistemas auto-distributivos e serverless computing

Alexandre Ladeira Campanhã da Silva * Gabriel Pallotta Cardenete *
Luiz Fernando Bittencourt * Roberto Rodrigues Filho †

Resumo

Este projeto analisa e compara os desafios e impactos da adoção de sistemas auto-distributivos e serverless computing em ambientes de computação em nuvem, especificamente avaliando o desempenho de sistemas executados no Google Cloud Run em comparação com sistemas implementados em Kubernetes usando o modelo SDS (Self Distributing Systems). Baseando-se em pesquisas anteriores, o estudo busca responder questões sobre diferenças de desempenho, identificar cenários específicos de vantagem para cada abordagem e avaliar suas características em termos de uso de recursos e tempo de resposta. Foram conduzidos testes de desempenho detalhados nos sistemas implementados, submetendo-os a diversas cargas de trabalho e monitorando métricas críticas. Os resultados fornecem uma visão abrangente das vantagens e desvantagens dos sistemas auto-distributivos e serverless computing em ambientes de computação em nuvem.

1 Introdução

O crescente avanço da complexidade na área de desenvolvimento de software torna cada vez mais crucial a busca por ferramentas que facilitam a gestão de sistemas, especialmente permitindo que autonomicamente tomem decisões e se adaptem, evitando a necessidade de intervenções dos desenvolvedores [1].

A necessidade de adaptabilidade ganha destaque à medida que a heterogeneidade dos sistemas aumenta e a volatilidade nos ambientes operacionais se intensifica [2]. A capacidade de adicionar e remover nós de maneira eficiente torna-se crucial para garantir a flexibilidade desses sistemas diante das demandas dinâmicas. A busca por soluções capazes de se adaptar a essas mudanças sem comprometer a estabilidade e a eficiência se faz cada vez mais relevante.

Nesse cenário, esse projeto se propõe a analisar duas alternativas para realizar a adaptação autônoma de sistemas. A primeira utilizando o modelo serverless, com a ferramenta Google Cloud Run [15], e a segunda utilizando o modelo de sistemas auto-distributivos, desenvolvido em trabalhos anteriores, com a ferramenta Kubernetes [14].

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP
†Departamento de Computação, Universidade Federal de Santa Catarina, 88900-000 Araranguá, SC

Foram desenvolvidas duas aplicações com características distintas em relação à utilização de recursos, padrões de acesso e tráfego de rede. Assim, foram executados diversos testes de desempenho comparando ambas as abordagens, considerando métricas como tempo de resposta, número de requisições por segundo e uso de recursos.

Este relatório técnico é dividido da seguinte maneira:

- **Seção 2**: discute os principais conceitos teóricos para o desenvolvimento do trabalho.
- **Seção 3**: apresenta os objetivos do projeto e suas questões principais.
- **Seção 4**: discorre sobre como o trabalho foi elaborado para responder as perguntas da seção anterior.
- **Seção 5**: demonstra os resultados dos experimentos organizados na seção 4.
- **Seção 6**: comenta sobre possíveis trabalhos futuros que podem se aproveitar dos resultados encontrados no projeto.
- **Seção 7**: apresenta a conclusão e considerações finais.

2 Referencial Teórico

Nesta seção, são apresentados os fundamentos teóricos relacionados ao trabalho. São abordados os conceitos de computação autônoma, sistemas auto-adaptativos, arquiteturas serverless, como o Google Cloud Run, e é discutida a orquestração de *containers*, incluindo tecnologias como Kubernetes. Essas plataformas e abordagens são consideradas no contexto da implementação e gestão de sistemas auto-adaptativos, enriquecendo a compreensão das práticas atuais e suas interseções com os sistemas emergentes.

2.1 Computação Autonômica e Sistemas Auto-adaptativos

Segundo Kephart e Chess, que realizam uma análise de um manifesto da IBM publicado em 2001 [1], existe uma crise na área de desenvolvimento de software, causada por uma crescente complexidade nos sistemas desenvolvidos. Nesse cenário, aplicações chegam a milhões e até mesmo dezenas de milhões de linhas de código, tornando-se cada vez mais difícil a manutenção e evolução dos sistemas já existentes.

Uma possível solução discutida por Kephart e Chess seria a utilização da computação autonômica, que envolve a criação de sistemas com a capacidade de se auto-gerenciarem, a partir de configurações especificadas pelos desenvolvedores. Esses sistemas seriam capazes de se auto-configurar, auto-otimizar, auto-curar e auto-proteger, sem a necessidade de intervenção manual dos desenvolvedores.

Kephart e Chess detalham esses aspectos, mostrando como a computação autonômica traria melhorias se comparada à computação tradicional:

- **Auto-configuração**: configuração automatizada dos componentes e sistemas seguindo políticas de alto nível, com ajustes automáticos e ininterruptos. Com isso, temos uma economia de tempo para realizar essas configurações, e erros humanos são evitados.

- **Auto-otimização**: componentes e sistemas continuamente buscam formas de melhorar sua performance e eficiência, evitando a complexidade de gerenciar o *fine-tuning* de parâmetros de execução manualmente.

- **Auto-cura**: os sistemas automaticamente detectam, realizam o diagnóstico e reparam problemas de software e hardware, evitando o desperdício de tempo de desenvolvedores buscando e solucionando problemas em sistemas complexos.

- **Auto-proteção**: sistemas automaticamente se defendem de ataques maliciosos ou falhas em cascata, utilizando de alertas para antecipar e prevenir falhas gerais.

Esse projeto explora principalmente o aspecto de auto-otimização, analisando sistemas desenvolvidos com ferramentas que permitem uma adaptação autônoma em tempo de execução, seja na escala do sistema (número de instâncias), ou na composição interna do próprio.

2.2 Sistemas de Software Emergentes

Segundo Filho [5], Sistemas de Software Emergentes são definidos como sistemas construídos com componentes pequenos e reutilizáveis, que permitem auto-composição e auto-otimização em tempo de execução.

Além disso, é descrito o *framework* PAL (*Perception*, *Assembly*, *Learning*), que apresenta uma arquitetura que permite a construção de tais sistemas, e é composto por três módulos:

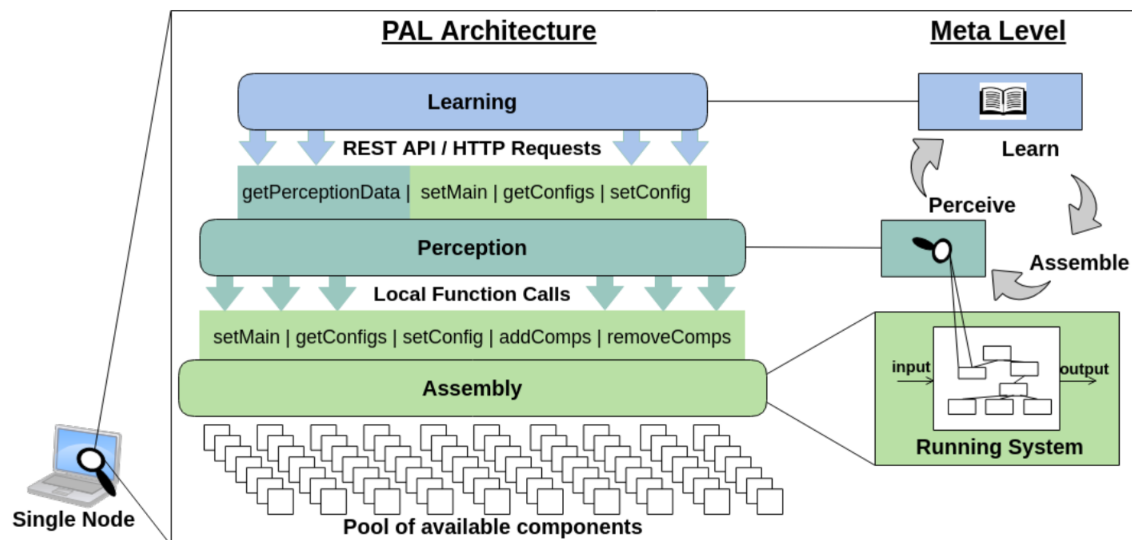

Figura 1: Arquitetura do *framework* PAL [5]

- **Assembly**: módulo responsável por compor e gerenciar a estrutura do sistema. Ele cria e supervisiona a interação entre os componentes do sistema, permitindo a modificação, adição ou remoção de elementos conforme necessário. Além disso, o módulo

facilita a descrição e a execução de composições de software por meio de suas funcionalidades.

- **Perception**: usa o módulo de *Assembly* para monitorar o sistema e o ambiente, oferecendo suas funções via REST API para facilitar o acesso a processos de aprendizado e descrevendo dados de percepção coletados no ambiente de execução.
- **Learning**: utiliza os módulos de *Perception* e *Assembly*, compreendendo as relações entre os componentes e o desempenho do sistema, controlando o aprendizado ativo a partir de objetivos pré-estabelecidos e explorando e ajustando composições para diferentes condições.

2.3 Arquiteturas de Serverless Computing

As arquiteturas de serverless computing incorporam serviços de terceiros *"Backend as a Service"* (BaaS) e/ou que incluem código personalizado executado em *containers* gerenciados e efêmeros em uma plataforma *"Functions as a Service"* (FaaS). Essas arquiteturas removem grande parte da necessidade de um componente de servidor tradicional sempre ativo, o que pode resultar em uma redução significativa no custo operacional, complexidade e tempo de liderança de engenharia [8].

2.4 Google Cloud Run

O Google Cloud Run [15] é uma plataforma de computação totalmente gerenciada para implantar e dimensionar *containers* HTTP serverless, sem se preocupar com a configuração de *clusters*, provisionamento de máquinas ou dimensionamento automático [9].

2.5 Orquestração de *containers*

A orquestração de *containers* é um processo de gerenciamento automatizado que inclui o provisionamento, a implantação, o escalonamento e o monitoramento de vários *containers* [10, 11]. Ela facilita todas as etapas de gerenciamento, implantação, rede e escala dos mesmos [12]. A orquestração de *containers* é utilizada para automatizar grande parte do esforço operacional para executar serviços e fluxos de trabalho organizados em *containers* [10].

2.6 Kubernetes

O Kubernetes [14] é uma ferramenta *open source* de orquestração de *containers* projetada e desenvolvida originalmente por engenheiros do Google. Com sua capacidade de orquestração, é possível criar serviços de aplicações que abrangem múltiplos *containers*, programar uso desses *containers* em um *cluster*, escalá-los e gerenciar a integridade deles ao longo do tempo. O Kubernetes elimina grande parte dos processos manuais necessários para implantar e escalar aplicações [10].

3 Objetivos

Este projeto tem como objetivo analisar e comparar os desafios e impactos da adoção de sistemas auto-distributivos e serverless computing em ambientes de computação em nuvem, especificamente, avaliando o desempenho de um sistema executado no Google Cloud Run em comparação com um sistema implementado em Kubernetes utilizando o modelo SDS (Self Distributing Systems). Esta pesquisa se baseia em estudos anteriores, notadamente no relatório intitulado "Um Estudo sobre Sistemas Auto-distributivos em Ambientes Elásticos"e no artigo "A Self-distributing System Framework for the Computing Continuum."[4, 6].

A análise visa responder às seguintes questões:

1. Qual é a diferença de desempenho entre sistemas auto-distributivos e serverless computing, considerando variáveis como tempo de resposta, quantidade de requisições por segundo e utilização de recursos?

2. Em que cenários específicos os sistemas auto-distributivos superam serverless computing e vice-versa?

Para abordar essas questões, foram realizados testes de desempenho e análises detalhadas dos sistemas implementados no Google Cloud Run e no Kubernetes com SDS. Os sistemas foram submetidos a cargas de trabalho variadas, e métricas críticas foram monitoradas e registradas.

Os resultados deste estudo comparativo fornecerão uma visão abrangente sobre as vantagens e desvantagens de sistemas auto-distributivos e serverless computing em ambientes de computação em nuvem.

4 Metodologia

4.1 Framework SDS

Para explorar o modelo SDS, foi utilizado como base o trabalho de Oswaldo [3], que criou um ferramental na linguagem de programação Dana [13], que permite implementar a auto-distribuição para uma aplicação qualquer em Dana. Além disso, também foi utilizado como base os trabalhos de Dias [7] e Oliveira [4], que estenderam esse sistema com ferramentas que permitem a implantação e auto-distribuição no ambiente Kubernetes.

A estrutura do sistema pode ser vista na Figura 2. Abaixo são listados os componentes desse sistema e suas respectivas funções.

- *Distributor*: aplicação em Dana, responsável por receber as requisições e decidir como serão processadas (utilizar uma implementação local ou um *proxy*). Também é responsável por controlar a distribuição atual do sistema, utilizando o *ServerCTL* para implantar instâncias do *Remote Distributor*, e interagindo diretamente com elas.

- *ServerCTL*: aplicação em Python com uma API HTTP simples que integra com a API do Kubernetes e permite a criação e destruição de *deployments* do *Remote Distributor*

- *Remote Distributor*: aplicação em Dana que inicialmente executa sem nenhum módulo carregado, e pode ser configurada dinamicamente para executar certos módulos e receber requisições do *Distributor* dependendo da distribuição do sistema.

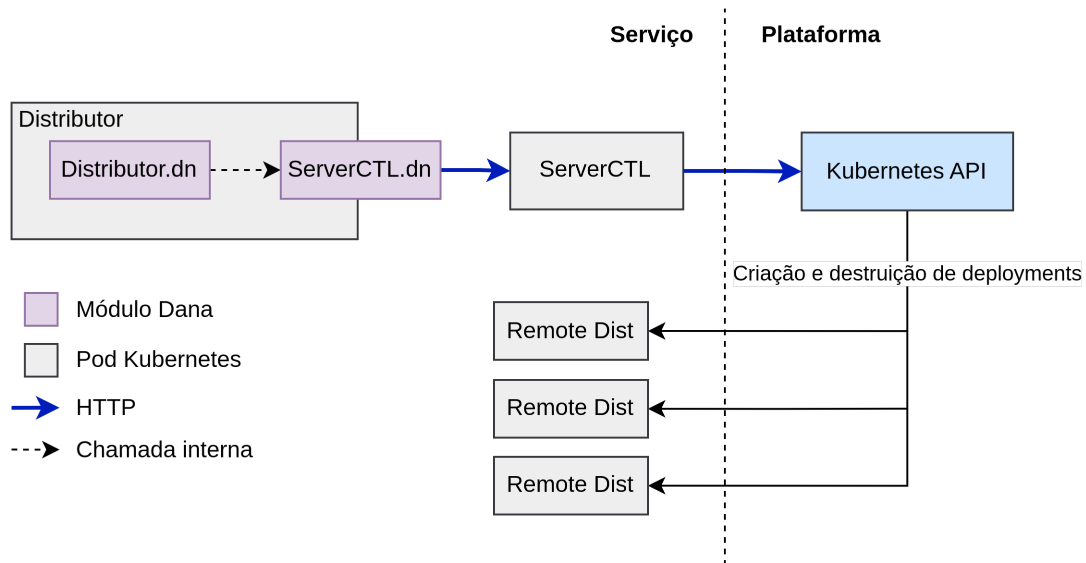

Figura 2: Arquitetura geral do sistema desenvolvido com o modelo SDS

4.2 Aplicação de teste de primalidade

Foi implementada uma aplicação simples de teste de primalidade, com o propósito de simular uma carga de trabalho com alto uso de CPU. O Algoritmo 1 mostra o pseudocódigo do algoritmo implementado, que propositalmente não conta com grandes otimizações para utilizar mais processamento.

```
1 if num <= 3 then
2     return num > 1;
3 for i ← start to end do
4     if num % i = 0 then
5         return false;
6 return true;
```

Algoritmo 1: Algoritmo simples de teste de primalidade

Esse algoritmo pode ser executado em um servidor, e utilizado para checar se um número n é primo utilizando os parâmetros $start = 2$ e $end = n$. Foram propostas duas formas de escalar esse algoritmo para execução em mais de uma máquina.

A primeira consiste em realizar uma distribuição das requisições entre as diversas máquinas, com um algoritmo de escalonamento como o Round-Robin.

Alternativamente, é possível dividir uma requisição em partes menores, e redirecionar cada parte para uma máquina diferente. Com a linguagem Dana [13] e o *framework* de SDS desenvolvido, é possível implementar essa distribuição de forma simples com um algoritmo que irá atuar em um *proxy* na frente dos nós remotos. O Algoritmo 2 mostra uma possível forma de implementar essa distribuição, com a divisão de intervalos iguais entre 0 a num, que serão enviados para cada nó remoto utilizando os parâmetros *start* e *end*. Cada nó remoto por sua vez, irá executar o algoritmo mostrado anteriormente.

```
1 for i ← 0 to length(remotes) do
2     rangeSize ← (num − 2)/length(remotes);
3     rangeStart ← 2 + i ∗ rangeSize;
4     rangeEnd ← 2 + (i + 1) ∗ rangeSize;
5     threads[i] ← makeRequestAsync(remotes[i], num, rangeStart, rangeEnd);
6 for i ← 0 to length(remotes) do
7     join(threads[i]);
```

Algoritmo 2: Algoritmo de distribuição de uma requisição em vários nós

No Algoritmo 2 também é relevante destacar que as requisições para nós remotos é feita de forma assíncrona, de forma que todos os nós remotos possam processar a requisição simultaneamente.

4.3 Aplicação de lista de usuários

A segunda implementação realizada foi uma aplicação simples de lista de usuários, com o propósito de simular uma carga de trabalho com alto uso de I/O e elevado uso de rede. Ela recebe uma lista de *ids* e retorna todas as informações dos usuários com os *ids* correspondentes armazenados na memória. Como essa aplicação foi desenvolvida para analisar intensidade de rede, se um *id* estiver presente n vezes na requisição, o usuário com o respectivo *id* estará n vezes na resposta.

O Algoritmo 3 mostra a implementação do processamento de uma requisição contendo diversos *ids*, e a resposta conterá todos os usuários.

```
1 for i ← 0 to length(userIds) do
2     id ← userIds[i];
3     response ← response + users.get(id);
4 return response;
```

Algoritmo 3: Algoritmo para retornar todos os usuários a partir de uma lista de ids

Para executar essa aplicação em mais de uma instância, é possível utilizar um algoritmo simples para distribuir as requisições de forma aleatória entre os nós remotos. Como cada instância manterá a lista de usuários em memória, e essa lista é imutável, não é necessário

realizar nenhuma lógica de sincronização de estados, sendo que cada instância consegue processar cada requisição independentemente. O Algoritmo 4 mostra uma possível forma de realizar essa implementação, utilizando uma função que gera números aleatórios.

1 $remote \leftarrow random(remotes);$
2 $response \leftarrow makeRequest(remote, ids);$

Algoritmo 4: Algoritmo de distribuição de requisições para nós remotos aleatórios

4.4 Ambiente de Execução

Para a execução das aplicações descritas anteriormente, foram utilizados dois ambientes distintos na plataforma Google Cloud [16]. O serviço Google Kubernetes Engine [17] foi utilizado para a criação de um *cluster* Kubernetes, que permitiu a execução dos componentes no modelo SDS. Para a execução do modelo serverless, foi utilizado o serviço Google Cloud Run [15]. Ambos os serviços foram configurados na mesma região, de forma a reduzir discrepâncias de latência entre eles.

4.4.1 Google Cloud Run

No ambiente do Google Cloud Run, as duas aplicações seguem a mesma arquitetura que pode ser vista na Figura 3, com a única diferença sendo o módulo Dana que contém a implementação específica para cada aplicação.

Dentro de cada instância, temos o módulo *Server*, que irá chamar o módulo da respectiva aplicação implementada (*Prime* para teste de primalidade e *Users* para lista de usuários), sem nenhum tipo de *proxy*. O Cloud Run gerencia as instâncias da aplicação automaticamente, bem como a forma como é feita a distribuição das requisições entre elas, por meio de um algoritmo de balanceamento de requisições.

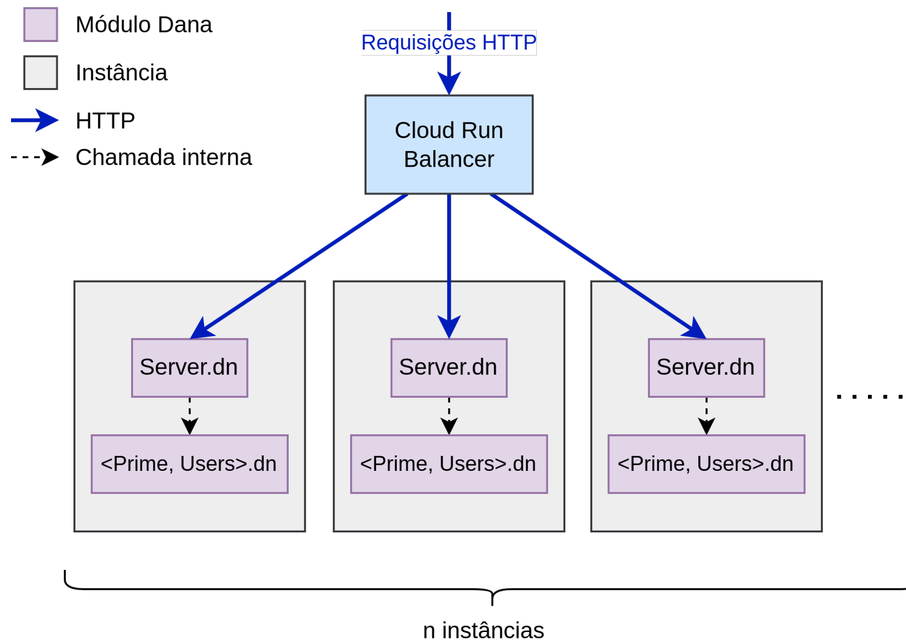

Figura 3: Organização de componentes no ambiente Cloud Run

Nesse cenário, foram configuradas na interface do Cloud Run apenas as especificações de cada instância da imagem e o número máximo de instâncias. Foram configurados 100 *millicpus* e 256 *mebibytes* de memória para cada instância, e o número máximo de instâncias foi definido como 4.

Como a CPU definida é menor do que 1 núcleo, o Cloud Run não permite um paralelismo maior do que 1 para cada instância. Portanto, todos os testes realizados possuem 4 ou menos *threads* de requisição, de forma a não serem impactados por essa limitação.

4.4.2 Google Kubernetes Engine

No ambiente do Google Kubernetes Engine (GKE), também temos a mesma arquitetura para ambas as aplicações, que é mostrada na Figura 4. A aplicação *Distributor* é responsável por receber as requisições HTTP e utiliza a implementação do módulo da aplicação (*Prime* ou *Users*) que pode ser carregada como um *proxy* que redireciona a requisição em múltiplas partes. Para o módulo *Prime*, essa distribuição é realizada com a lógica descrita no Algoritmo 2. Já para o módulo *Users*, é utilizada a lógica descrita no Algoritmo 4.

As múltiplas instâncias da aplicação *RemoteDist* são instanciadas por meio do componente ServerCTL, como descrito na seção 4.1. Cada instância receberá as requisições e processará-las usando o módulo padrão de cada aplicação, implementando seus respectivos algoritmos.

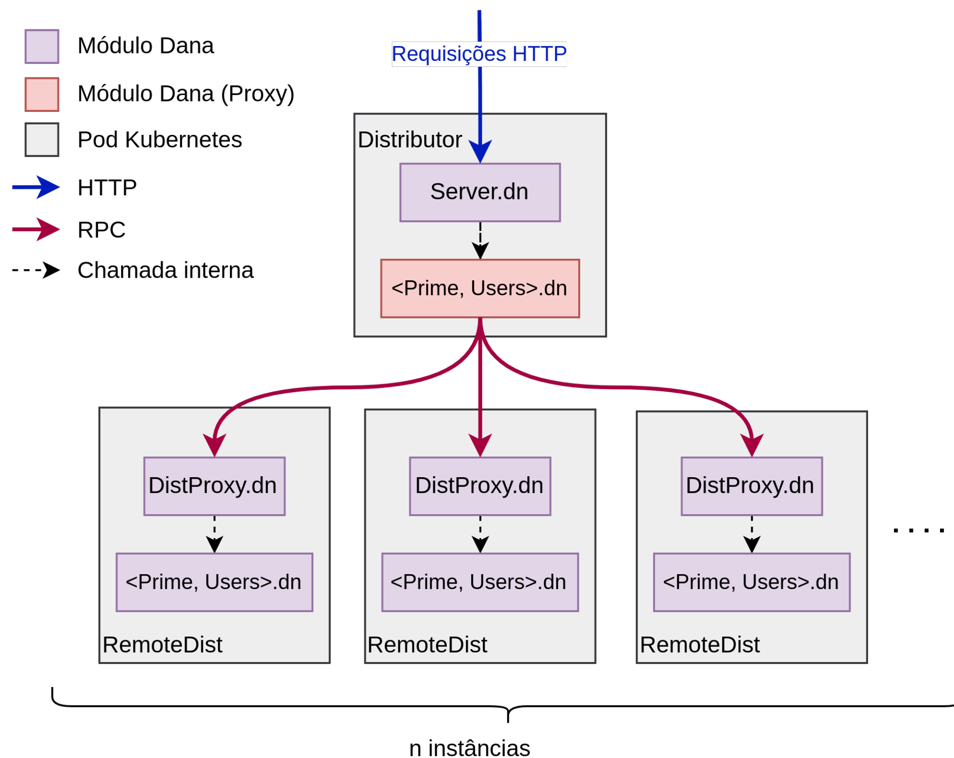

Figura 4: Organização de componentes no ambiente Kubernetes

A instância do *Distributor* foi configurada com 400 *millicpus* e 256 *mebibytes* de memória. Para cada instância do *RemoteDist* foram configurados 100 *millicpus* e 256 *mebibytes* de memória. Dessa forma, podemos ter uma possibilidade de configuração local da aplicação (sem *proxy*) com a mesma quantidade de recurso de CPU presentes em uma distribuição com 4 instâncias do *RemoteDist*.

Não foram utilizados processos de dimensionamento dinâmicos para a aplicação do *RemoteDist*, o número de instâncias foi redimensionado manualmente para 4 durante os testes realizados com a aplicação distribuída. Para testes realizados sem distribuição, a aplicação também foi manualmente configurada para não criar nenhuma instância do *RemoteDist*, e não carregar o módulo *proxy*.

O *cluster* do GKE foi criado sem a opção *autopilot*, para que não fossem realizadas alterações nos nós do *cluster* durante a execução dos testes, prejudicando os resultados obtidos. Mesmo assim, não é possível ter controle sobre qual nó do *cluster* cada recurso é alocado, e por isso os valores de recursos de CPU e memória para cada aplicação foram definidos sempre utilizando os valores limites, de forma a garantir que cada aplicação é alocada em um nó com recursos suficientes para atender à sua demanda.

4.5 Ferramentas de Teste

Foram utilizadas duas ferramentas principais para executar os testes e extrair métricas: Grafana k6 [18] e Prometheus [19].

O k6 permite a execução de testes de carga sobre um aplicação, com a possibilidade de configuração por meio de um *script* simples na linguagem JavaScript.

Para a aplicação de teste de primalidade, foi criado o *script* exibido na Figura 7, que realiza requisições POST do protocolo HTTP para uma aplicação executando em uma URL, com o corpo da requisição sendo um número cuja primalidade será verificada. O k6 permite a configuração de número de *threads* de requisições, pelo parâmetro *virtual users* (VUs). Cada *thread* realiza requisições continuamente sem interrupções, somente sendo limitadas pelo tempo de resposta da aplicação.

```
import http from 'k6/http';
import { check } from 'k6';

export default function () {
  const url = '<url>';
  const primeNumber = '10007';
  const res = http.post(url, primeNumber);

  check(res, {
    'is status 200': (r) => r.status === 200,
    'is prime': (r) => r.body == "PRIME",
  });
}
```

Figura 5: Código utilizando k6 para executar múltiplas requisições para a aplicação de teste de primalidade

Para a aplicação de lista de usuários, foi criado um *script* semelhante, apresentado na Figura 6, que permite a criação de uma lista de n *ids* que serão incluídos na requisição.

```
import http from 'k6/http';
import { check } from 'k6';

export default function () {
  const url = '<url>'
  const n = 5 // (n * 100) ids

  const groups = Array.from({ length: 100 }, (_, i) => i);
  const ids = Array.from({ length: n }, () => groups).flat().join(",");

  const res = http.post(url, ids);

  check(res, {
    'is status 200': (r) => r.status === 200,
  });
}
```

Figura 6: Código utilizando k6 para executar múltiplas requisições para a aplicação de lista de usuários

Com o k6 é possível exportar métricas dos testes de carga para a ferramenta Prometheus, que permite a visualização dessas métricas ao longo do tempo. Dessa forma, podemos analisar e extrair resultados como: tempo de resposta total, tempo de processamento da aplicação e número de requisições por segundo. Além disso, é possível utilizar a ferramenta para calcular os valores mínimos e máximos, médias e percentis ao longo de um intervalo de tempo.

Figura 7: Exemplo de gráfico gerado no Prometheus que permite a extração das métricas

Abaixo temos o comando para executar o k6, habilitando a exportação de métricas para o Prometheus.

```
# K6_PROMETHEUS_RW_TREND_STATS=count,sum,avg,p(90) k6 run --out \
experimental-prometheus-rw --vus 4 --duration 5m k6.js
```

Também foram utilizadas métricas disponíveis direto da plataforma Google Cloud. A principal métrica utilizada foi a de uso de CPU. Para a execução no Cloud Run, consideramos o uso de CPU reportado pela própria plataforma, considerando o número total de instâncias. Para o Kubernetes, consideramos o uso de CPU por *pod*, reportado pelo *cluster* Kubernetes.

4.6 Cenários de Teste

4.6.1 Aplicação de teste de primalidade

Para a definição dos diferentes casos de teste para a aplicação de teste de primalidade, variou-se o número de *threads* de requisição da ferramenta k6, e o número primo utilizado nas requisições.

Foram escolhidos 3 cenários de teste, com 1, 2 e 4 *threads* cada. Em cada cenário são considerados 4 números primos distintos: 5, 881, 10007 e 100109. Para cada caso, são realizadas requisições continuamente pela ferramenta k6 por 5 minutos, e então as métricas de desempenho são extraídas considerando a média nesse período.

4.6.2 Aplicação de lista de usuários

Para aplicação de lista de usuários, definimos 2 cenários de teste.

No primeiro cenário foram utilizadas requisições com uma lista pequena de usuários, apenas 500, com o intuito de observar o comportamento sem alta intensidade de rede. Já no segundo cenário, a lista utilizada possuía um número grande de usuários, 10000, para gerar alta intensidade de rede.

Em ambos os cenários, o ambiente SDS foi mantido com a configuração local (sem distribuição), como descrito na seção 4.4.2, para que esse ambiente pudesse ser analisado contra uma abordagem distribuída (serverless), com o objetivo de identificar as vantagens e desvantagens de realizar a distribuição de requisições em cenários com diferentes intensidades de rede.

Assim como descrito na seção anterior, também foram realizados requisições continuamente por 5 minutos, extraindo as métricas considerando a média nesse período.

5 Resultados

Nas seguintes seções são apresentados os resultados de execução para as aplicações e cenários de testes propostos anteriormente. Em seguida, são realizadas algumas discussões gerais sobre os resultados obtidos.

5.1 Aplicação de teste de primalidade

5.1.1 Cenário com 1 *thread*

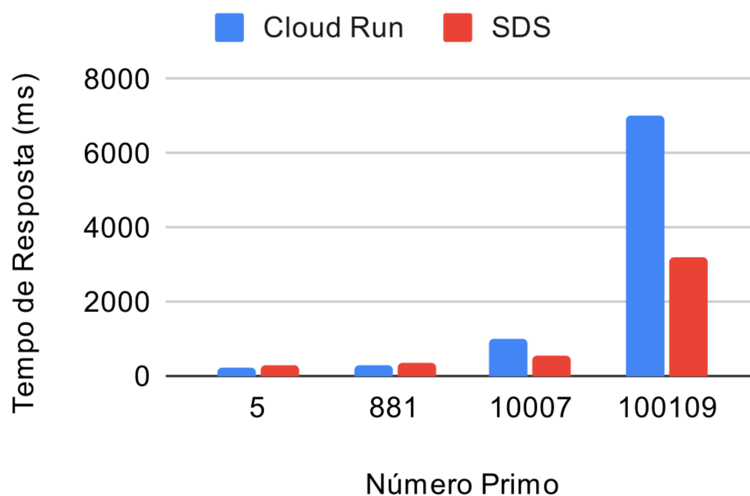

Figura 8: Tempo de resposta por número primo em cada ambiente, com 1 *thread* de requisição

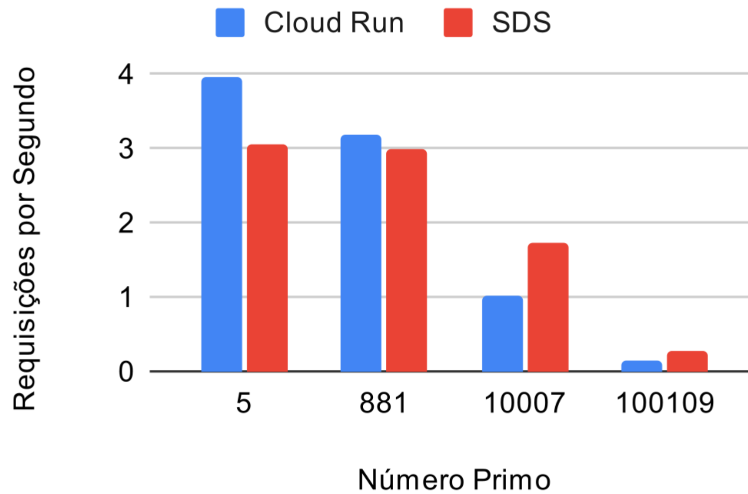

Figura 9: Requisições por segundo por número primo em cada ambiente, com 1 *thread* de requisição

Nos gráficos das Figuras 8 e 9 temos os tempos de resposta e requisições por segundo obtidas para cada caso de teste, no cenário de somente uma *thread* de requisição no k6.

Para esse cenário, esperava-se que o uso de recursos do Cloud Run fosse menor, devido à baixa concorrência, enquanto que o SDS teria uma utilização melhor de recursos, levando a um tempo menor de processamento, e consequentemente um maior número de requisições atendidas por segundo.

Isso de fato foi observado, exceto nos casos "5"e "881", nos quais o SDS teve um desempenho inferior ao Cloud Run, tanto na métrica de tempo de resposta, como em requisições por segundo.

Durante os testes, foi possível notar que o ambiente do Cloud Run conseguiu estabelecer a conexão mais rapidamente do que no ambiente SDS, provavelmente por otimizações que a plataforma do Cloud Run aplica automaticamente. Esse tempo a mais é relevante apenas para cargas pequenas, onde o tempo de resposta total é menor, e explica o tempo de resposta maior do SDS nesses cenários.

5.1.2 Cenário com 2 *threads*

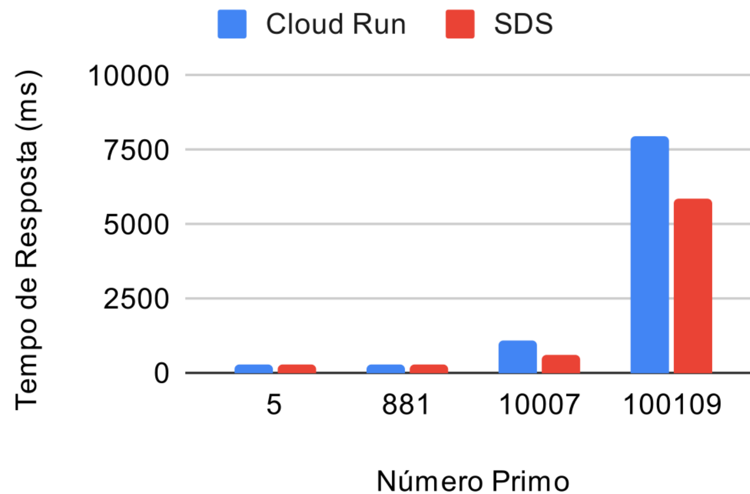

Figura 10: Tempo de resposta por número primo em cada ambiente, com 2 *threads* de requisição

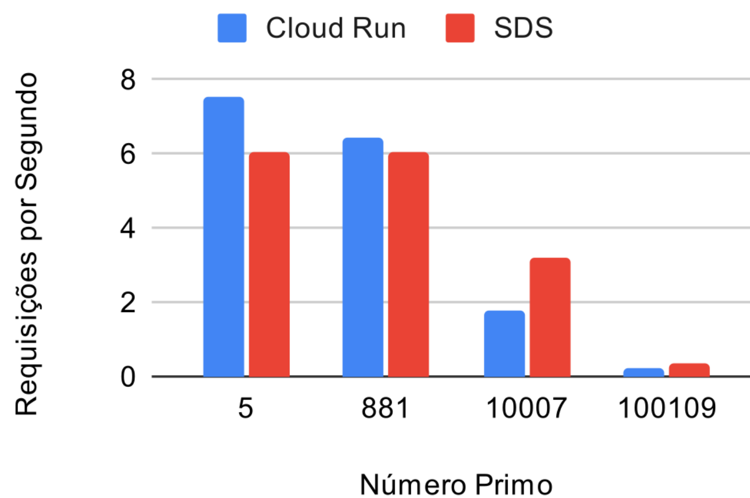

Figura 11: Requisições por segundo por número primo em cada ambiente, com 2 *threads* de requisição

Nos gráficos das Figuras 10 e 11 considera-se o cenário de 2 *threads* de requisição no k6.

Assim como no cenário anterior, observa-se um desempenho superior do SDS em relação ao Cloud Run, exceto nos cenários "5"e "881", que são mais impactados pela mesma latência de conexão adicional descrita anteriormente.

Porém, a vantagem do SDS é menor nesse cenário, já que a utilização de paralelismo do Cloud Run é maior em relação ao cenário anterior.

Ainda em comparação com o cenário anterior, tanto o Cloud Run como o SDS apresentam aproximadamente o dobro de requisições por segundo, o que é esperado pelo fato de termos o dobro de *threads* de requisição. Porém, esse número não é exatamente o dobro devido aos *overheads* de concorrência esperados de ambos os ambientes, que causam tempos de resposta ligeiramente superiores.

5.1.3 Cenário com 4 *threads*

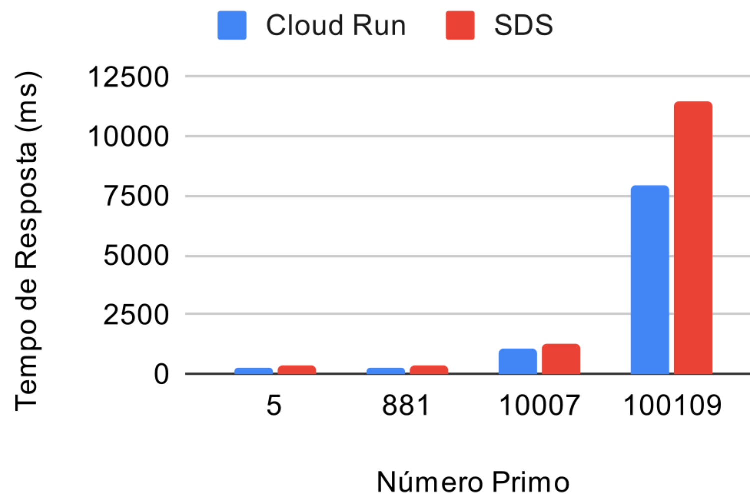

Figura 12: Tempo de resposta por número primo em cada ambiente, com 4 *threads* de requisição

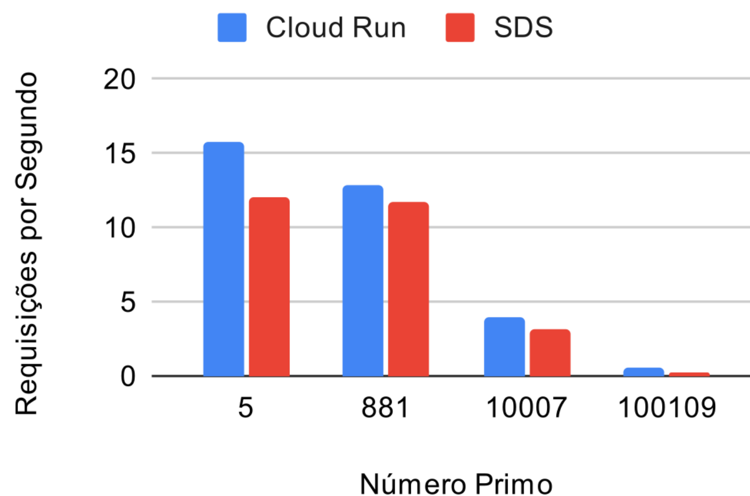

Figura 13: Requisições por segundo por número primo em cada ambiente, com 4 *threads* de requisição

Finalmente, nos gráficos das Figuras 12 e 13 analisa-se o cenário de 4 *threads* de requisição no k6.

Nesse cenário, nota-se que o desempenho do SDS é inferior ao serverless em todos os casos de teste. Para os casos com números primos menores, essa diferença é menos perceptível, mas aumenta com os casos maiores.

Tanto o Cloud Run como o SDS utilizam os seus paralelismos a um nível máximo nesse cenário, já que ambos estão limitados a 4 instâncias de concorrência, e não há uma subutilização de recursos por parte do Cloud Run.

O que causa o desempenho inferior do SDS nesse caso é o *overhead* do mecanismo de *sharding* implementado, que realiza mais operações no total se comparado ao algoritmo simples que executa em cada instância do Cloud Run.

5.1.4 Análise de uso de recursos

Uma métrica que permite um entendimento melhor acerca da diferença de desempenho entre o SDS e Cloud Run nos cenários apresentados é o uso de CPU médio, apresentado para cada caso de teste nos gráficos das Figuras 14 e 15.

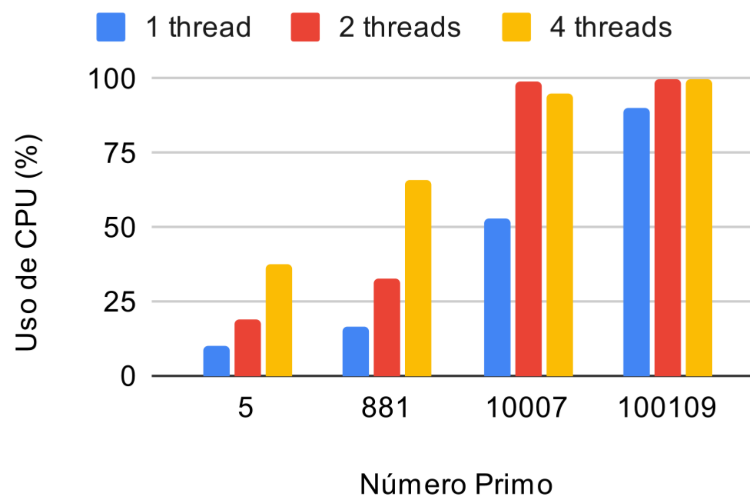

Figura 14: Uso de CPU por número primo no ambiente em SDS para cada caso de teste e número de *threads*

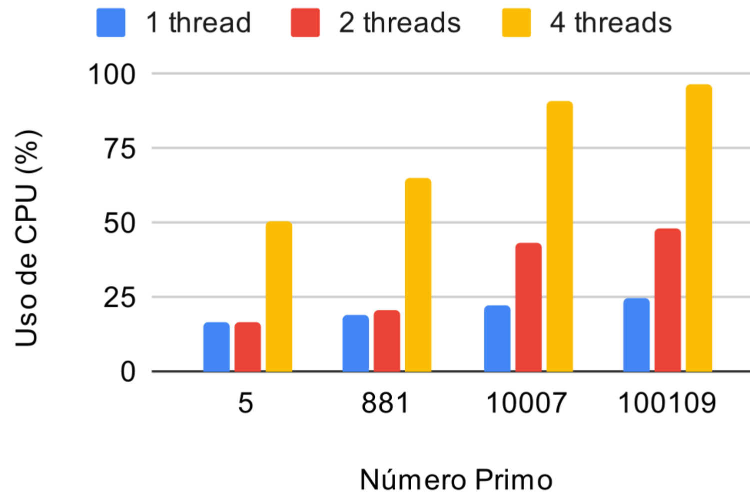

Figura 15: Uso de CPU por número primo no ambiente em Cloud Run para cada caso de teste e número de *threads*

Nota-se que o uso de CPU do SDS é maior do que o Cloud Run, especialmente nos cenários de 1 e 2 *threads*. Isso justifica o melhor desempenho obtido pelo SDS nesses cenários.

Observa-se que no ambiente de orquestração de *containers* do Cloud Run a quantidade de *pods* em execução foi reduzida automaticamente em função do número de solicitações simultâneas. Essa abordagem pode levar a uma utilização mais eficiente de recursos, uma vez que menos *pods* são provisionados. Porém, essa otimização pode ter como contrapartida uma diminuição na velocidade de execução dos processos, causando um maior tempo de resposta.

Já nos testes executados em Kubernetes utilizando SDS, foi alocado um número fixo de *pods* para atender as solicitações. Nesse cenário, a velocidade das solicitações pôde ser otimizada dividindo seu processamento entre as máquinas disponíveis, gerando tempos de requisição menores. No entanto, essa abordagem pode levar a uma utilização menos eficiente dos recursos em cenários como o de números primos menores, uma vez que todos os *pods* estão sempre em execução, independentemente da carga de trabalho.

5.2 Aplicação de lista de usuários

5.2.1 Análise de uso de tempo de resposta

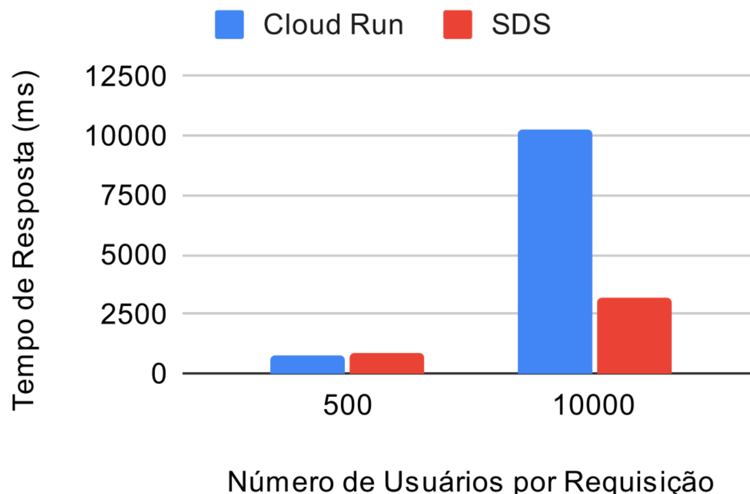

Figura 16: Tempo de resposta por cenário de teste, para cada ambiente

No gráfico da Figura 16 temos os tempos de resposta obtidos para cada cenário de teste descritos na seção 4.6.2.

No cenário com 500 usuários, como as requisições executadas não exigiam grande intensidade de rede, percebe-se que o desempenho dos dois ambientes é similar.

Já no cenário com 10000 usuários, o ambiente distribuído do Cloud Run acaba apresentando um desempenho quase quatro vezes menor que o SDS. Como o ambiente serverless precisa distribuir a chamada em quatro instâncias diferentes, um *overhead* de rede e uso de I/O acaba sendo criado afetando o tempo de resposta significativamente, enquanto o ambiente SDS se aproveita da configuração local para evitar o mesmo problema.

5.2.2 Análise de uso de recursos

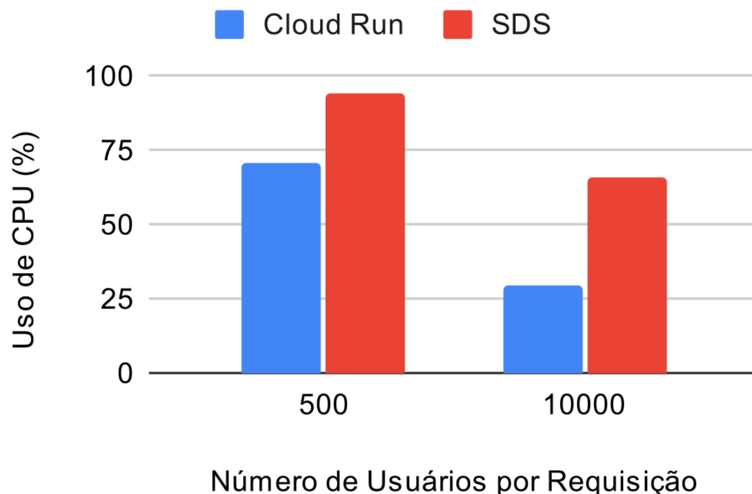

Figura 17: Uso de CPU por cenário de teste, para cada ambiente

No gráfico da Figura 17 observamos a comparação de uso de CPU para os mesmo cenários discutidos anteriormente.

No cenário com 500 usuários, ambos os ambientes possuem um uso de recursos mais próximos, já que nesse cenário, o gargalo de rede não é tão significativo.

Já no cenário com 10000 usuários, pode-se notar que a utilização de CPU do SDS é mais do que duas vezes maior do que o Cloud Run. Isso se deve à sua configuração local, que utiliza menos I/O e rede, conseguindo obter uma utilização melhor de processamento, e como consequência, tempos de resposta menores, como visto no gráfico da Figura 16.

6 Trabalhos Futuros

Nessa seção, são discutidas limitações desse projeto, que podem servir como oportunidades para trabalhos futuros.

No estudo não foram utilizadas técnicas para realizar a alocação dinâmica nos ambientes SDS. Até o momento, os testes executados foram limitados a uma configuração estática de pods, deixando de lado a dinamicidade que a alocação poderia oferecer, especialmente ao empregar técnicas de aprendizado de máquina para otimizar a alocação de recursos com base nas demandas em tempo real.

Além disso, para ampliar a diversidade de cenários avaliados, os testes de carga poderiam considerar diferentes origens de clientes, simulando máquinas com configurações distintas e em redes diferentes. Essa abordagem permitiria uma análise mais abrangente sobre como os sistemas respondem a diferentes fontes de carga.

Devido à limitações do tamanho do *cluster* Kubernetes criado para esse projeto, o número de instâncias e uso de recursos ficou limitado. A realização de testes com um número maior

de réplicas e a otimização de recursos poderiam fornecer *insights* valiosos sobre escalabilidade e pontos críticos do sistema.

Por fim, seria possível ampliar o escopo das aplicações testadas, incluindo aquelas com outras características, como as voltadas para intensidade de uso de memória. Esses outros cenários ofereceriam uma compreensão ainda mais profunda de como diferentes tipos de carga impactam as ferramentas em estudo, fornecendo informações essenciais para casos de uso específicos.

7 Conclusão

O estudo comparativo revelou diferenças no desempenho entre sistemas auto-distributivos e serverless computing, especialmente no que diz respeito a variáveis como tempo de resposta, quantidade de requisições por segundo e utilização de recursos. Já o serverless computing, exemplificado pelo Cloud Run, destaca-se pela gestão automática da distribuição em nível de plataforma, oferecendo simplicidade na implementação e uma camada de auto-otimização para a aplicação. No entanto, essa abordagem pode não ser ideal para certos cenários, como os de baixa concorrência ou alto uso de rede apresentados nesse estudo, que podem acabar resultando em um uso ineficiente de recursos.

Por outro lado, os sistemas auto-distributivos (SDS) proporcionam aos desenvolvedores um controle mais granular sobre a distribuição em nível de aplicação, permitindo ajustes precisos para uma variedade de cenários considerados durante o desenvolvimento. Isso permite os melhores resultados de desempenho vistos nos cenários de menor concorrência, e nos cenários com alto uso de I/O e rede, onde uma configuração local pode ser mais vantajosa do que distribuir a aplicação. No entanto, por ainda estarem em estágio inicial de desenvolvimento, essas ferramentas demandam maior complexidade na configuração para aplicações desenvolvidas, em comparação com a simplicidade oferecida pelo Cloud Run.

Quanto aos cenários específicos, os sistemas auto-distributivos se sobressaem em contextos que exigem ajustes precisos e uma adaptação específica da distribuição para diferentes demandas de aplicação. Por outro lado, o serverless computing apresenta-se como uma escolha mais conveniente para implementações rápidas, especialmente em situações de carga variável e quando a complexidade de configuração não é uma limitação.

Assim, o trabalho conseguiu promover a discussão planejada e responder as questões levantadas na seção de objetivos. A resposta à primeira questão, referente à diferença de desempenho entre os sistemas auto-distributivos e serverless computing, foi discutida detalhadamente durante a análise dos resultados na Seção 5. Já a segunda questão, sobre os cenários específicos em que cada abordagem se destaca, foi consolidada nos parágrafos anteriores.

Em conclusão, a escolha entre sistemas auto-distributivos e serverless computing depende das necessidades específicas do projeto. Ambas as abordagens têm seus pontos fortes e desafios, e a decisão deve considerar cuidadosamente as demandas de desempenho, flexibilidade e facilidade de implementação de cada cenário de aplicação.

Referências

[1] J. O. Kephart and D. M. Chess, The vision of autonomic computing, in C omputer, vol. 36, no. 1, pp. 41-50, Jan. 2003. https://doi.org/10.1109/MC.2003.1160055

[2] R. R. Filho and B. Porter, Autonomous State-Management Support in Distributed Self-adaptive Systems, 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C), 2020, pp. 176-181. https://doi.org/10.1109/ACSOS-C51401.2020.00052

[3] G. H. R. Oswaldo, L. F. Bittencourt, and R. R. Filho. Gestão transparente do estado para sistemas auto-distribuídos: primeiro estudo de caso, 2021, https://www.ic.unicamp.br/~reltech/PFG/2021/PFG-21-50.pdf

[4] A. P. Oliveira, R. H. Koaro, L. F. Bittencourt, and R. R. Filho. Um Estudo sobre Sistemas Auto-distributivos em Ambientes Elásticos, 2022, https://www.ic.unicamp.br/~reltech/PFG/2022/PFG-22-05.pdf

[5] R. Rodrigues Filho. Emergent Software Systems. PhD thesis, Lancaster University, 2018

[6] R. R. Filho, R. S. Dias, J. Seródio, B. Porter, F. M. Costa, E. Borin and L. F. Bittencourt. A Self-distributing System Framework for the Computing Continuum, 2023, https://robertovrf.github.io/pdfs/icccn2023rodriguesfilho.pdf

[7] R. S. Dias, R. R. Filho, L. F. Bittencourt and F. M. Costa. Runtime Microservice Self-distribution for Fine-grain Resource Allocation, 2022, https://robertovrf.github.io/pdfs/cloudam2022dias.pdf

[8] Baldini, I., et al. Serverless computing: Current trends and open problems. Research advances in cloud computing, 2017, 1-20. https://arxiv.org/abs/1706.03178

[9] Cloud Run: What no one tells you about Serverless (and how it's done) https://cloud.google.com/blog/topics/developers-practitioners/cloud-run-story-serverless-containers

[10] What is container orchestration? (Red Hat) https://www.redhat.com/en/topics/containers/what-is-container-orchestration

[11] What is container orchestration? (IBM) https://www.ibm.com/topics/container-orchestration

[12] Kubernetes: entenda a ferramenta de orquestração de containers https://www.microserviceit.com.br/kubernetes/

[13] Linguagem Dana. http://projectdana.com.

[14] Kubernetes https://kubernetes.io/

[15] Google Cloud Run https://cloud.google.com/run

[16] Google Cloud https://cloud.google.com/

[17] Google Kubernetes Engine https://cloud.google.com/kubernetes-engine

[18] Grafana k6 https://k6.io/

[19] Prometheus https://prometheus.io/

Apêndice

A Repositórios

- Repositório com o código implementado: `https://github.com/gabrielpallotta/pfg-alexandre-gabriel`